NETWORK INTERFACE AND PROTOCOL

BACKGROUND OF THE INVENTION

1. Field of the Invention

[0001] This invention relates to a network interface and a protocol for use in passing data over a network.

2. Description of the Related Art

[0002] When data is to be transferred between two devices over a data channel, each of the devices must have a suitable network interface to allow it to communicate across the channel. The devices and their network interfaces use a protocol to form the data that is transmitted over the channel, so that it can be decoded at the receiver. The data channel may be considered to be or to form part of a network, and additional devices may be connected to the network.

[0003] The Ethernet system is used for many networking applications. Gigabit Ethernet is a high-speed version of the Ethernet protocol, which is especially suitable for links that require a large amount of bandwidth, such as links between servers or between data processors in the same or different enclosures. Devices that are to communicate over the Ethernet system are equipped with network interfaces that are capable of supporting the physical and logical requirements of the Ethernet system. The physical hardware component of network interfaces are referred to as network interface cards (NICs), although they need not be in the form of cards: for instance they could be in the form of integrated circuits (ICs) and connectors fitted directly on to a motherboard.

[0004] Where data is to be transferred between cooperating processors in a network, it is common to implement a memory mapped system. In a memory mapped system communication between the applications is achieved by virtue of a portion of one

1

application's virtual address space being mapped over the network onto another application. The "holes" in the address space which form the mapping are termed apertures.

[0005] FIG. 1 illustrates a mapping of the virtual address space (X_o-X_n) onto another virtual address space (Y_o-Y_n) via a network. In such a system a CPU that has access to the X_o-X_n memory space could access a location x_1 for writing the contents of a register r_1 to that location by issuing the store instruction [st r_1, x_1]. A memory mapping unit (MMU) is employed to map the virtual memory onto physical memory location.

[0006] The following steps would then be taken:
1. CPU emits the contents of r_1 (say value 10) as a <u>write</u> operation to virtual address x_1
2. The MMU (which could be within the CPU) turns the virtual address x_1 into physical address pci_1 (this may include page table traversal or a page fault)
3. The CPU's write buffer emits the "write 10, pci_1" instruction which is "caught" by the controller for the bus on which the CPU is located, in this example a PCI (Input/Output bus subsystem) controller. The instruction is then forwarded onto the computer's PCI bus.
4. A NIC connected to the bus and interfacing to the network "catches" the PCI instruction and forwards the data to the destination computer at which virtual address space (Y_o-Y_n) is hosted.
5. At the destination computer, which is assumed to have equivalent hardware, the network card emits a PCI write transaction to store the data in memory
6. The receiving application has a virtual memory mapping onto the memory and may read the data by executing a "load Y_1" instruction

[0007] These steps are illustrated by FIG. 2. This figure illustrates that at each point that the hardware store instructions passes from one hardware device to another, a translation of the address from one address space to another may be required. Also note that a very

similar chain of events supports read operations and PCI is assumed but not required as the host IO bus implementation.

[0008] Hence the overall memory space mapping $\{X_o\text{-}X_n\} \rightarrow \{Y_o\text{-}Y_n\}$ is implemented by a series of sub-mappings as follows:

$\{X_o\text{-}X_n\}$

$\rightarrow$

$\{PCI_o, PCI_n\}$ (processor 1 address space)

$\rightarrow$

$\{PCI'_o, PCI'_n\}$ (PCI bus address space)

$\rightarrow$

Network--mapping not shown

$\rightarrow$

$\{PCI''_o\text{-}PCI''_n\}$ (destination PCI bus address space)

$\rightarrow$

$\{mem_o\text{-}mem_n\}$ (destination memory address space)

$\rightarrow$

$\{Y_o\text{-}Y_n\}$ (destination application's virtual address space)

[0009] The step marked in FIG. 2 as "Network" requires the NIC/network controller to forward the transaction to the correct destination host in such a way that the destination can continue the mapping chain. This is achieved by means of further memory apertures.

[0010] Two main reasons for the use of aperture mappings are:

[0011] a) System robustness. At each point that a mapping is made, hardware may check the validity of the address by matching against the valid aperture tables. This guards against malicious or malfunctioning devices. The amount of protection obtained is the ratio of the address space size (in bits) and the number of allocated apertures (total size in bits).

3

[0012] b) Conservation of address bits. Consider the case that a host within a 32 bit memory bus requires to access two 32 bit PCI buses. On the face of it, these would not be sufficient bits, since a minimum of 33 should be needed. However, the use of apertures enables a subset of both PCI buses to be accessed:

$$\{PCI_o\text{-}PCI_n\} \rightarrow \{PCI'_o\text{-}PCI'_n\} \text{ and}$$
$$\{PCI_p\text{-}PCI_q\} \rightarrow \{PCI''_p\text{-}PCI''_q\}$$

[0013] Hardware address mappings and apertures are well understood in the area of virtual memory and I/O bus mapping (e.g. PCI). However there are difficulties in implementing mappings are required over a network. The main issues are:

1. A large number of apertures may be required to be managed, because one host must communicate with many others over the network.

2. In network situations it is normal, for security reasons, to treat each host as its own protection and administrative domain. As a result, when a connection between two address spaces is to be made over a network, not all the aperture mappings along the path can be set up by the initiating host. Instead a protocol must be devised which allows all the mappings within each protection domain to be set by the "owner" of the domain in such a way that an end-end connection is established.

3. For security reasons it is normal for hosts within a network to not entirely trust the others, and so the mapping scheme should allow arbitrary faulty or malicious hosts from corrupting another host's memory.

[0014] Traditionally (e.g. for Ethernet or ATM switching), protocol stacks and network drivers have resided in the kernel. This has been done to enable

1. the complexity of network hardware to be hidden via a higher level interface;

2. the safe multiplexing of network hardware and other system resources (such as memory) over many applications;

3. the security of the system against faulty or malicious applications.

[0015] In the operation of a typical kernel stack system a hardware network interface card interfaces between a network and the kernel. In the kernel a device driver layer communicates directly with the NIC, and a protocol layer communicates with the system's application level.

[0016] The NIC stores pointers to buffers for incoming data supplied to the kernel and outgoing data to be applied to the network. These are termed the Rx data ring and the Tx data ring. The NIC updates a buffer pointer indicating the next data on the Rx buffer ring to be read by the kernel. The Tx data ring is supplied by direct memory access (DMA) and the NIC updates a buffer pointer indicating the outgoing data which has been transmitted. The NIC can signal to the kernel using interrupts.

[0017] Incoming data is picked off the Rx data ring by the kernel and is processed in turn. Out of band data is usually processed by the kernel itself. Data that is to go to an application-specific port is added by pointer to a buffer queue, specific to that port, which resides in the kernel's private address space.

[0018] The following steps occur during operation of the system for data reception:
1. During system initialization the operating system device driver creates kernel buffers and initializes the Rx ring of the NIC to point to these buffers. The OS also is informed of its IP host address from configuration scripts.
2. An application wishes to receive network packets and creates a Port which is a queue-like data structure residing within the operating system. It has a number port which is unique to host in such a way that network packets addressed by *<host:port>* can be delivered to the port's queue.
3. A packet arrives at the network interface card (NIC). The NIC copies the packet over the host I/O bus (e.g. a PCI bus) to the memory address pointed to by the next valid Rx DMA ring Pointer value.
4. Either if there are no remaining DMA pointers available, or on a pre-specified timeout, the NIC asserts the I/O bus interrupt in order to notify the host that data has been delivered.

5. In response to the interrupt, the device driver examines the buffer delivered and if it contains valid address information, such as a valid host address, passes a pointer to the buffer to the appropriate protocol stack (e.g. TCP/IP).

6. The protocol stack determines whether a valid destination port exists and if so, performs network protocol processing (e.g. generate an acknowledgement for the received data) and enqueues the packet on the port's queue.

7. The OS may indicate to the application (e.g. by rescheduling and setting bits in a "select" bit mask) that a packet has arrived on the network end point to which the port is bound. (By marking the application as runnable and invoking a scheduler).

8. The application requests data from the OS, e.g. by performing a recv() system call (supplying the address and size of a buffer) and whilst in the OS kernel, data is copied from the kernel buffer into the application's buffer. On return from the system call, the application may access the data from the application buffer.

9. After the copy, the kernel will return the kernel buffer to an O/S pool of free memory. Also, during the interrupt the device driver allocates a new buffer and adds a pointer to the DMA ring. In this manner there is a circulation of buffers from the free pool to an application's port queue and back again.

10. 10.An important property of the kernel buffers is that they are congruous in physical RAM and are never paged out by the VM system. However, the free pool may be shared as a common resource for all applications.

[0019] For data transmission, the following steps occur.

1. The operating system device driver creates kernel buffers for use for transmission and initializes the Tx ring of the NIC.

2. An application that is to transmit data stores that data in an application buffer and requests transmission by the OS, e.g. by performing a send() system call (supplying the address and size of the application buffer).

3. In response to the send() call, the OS kernel copies the data from the application buffer into the kernel buffer and applies the appropriate protocol stack (e.g. TCP/IP).

4. A pointer to the kernel buffer containing the data is placed in the next free slot on the Tx ring. If no slot is available, the buffer is queued in the kernel until the NIC indicates e.g. by interrupt that a slot has become available.

5. When the slot comes to be processed by the NIC it accesses the kernel buffer indicated by the contents of the slot by DMA cycles over the host IO bus and then transmits the data.

[0020] Considering the data movement through the system, it should be noted that in the case of data reception the copy of data from the kernel buffer into the application buffer actually leaves the data residing in the processor's cache and so even though there appears to be a superfluous copy using this mechanism, the copy actually serves as a cache load operation. In the case of data transmission, it is likely that the data that is to be transmitted originated from the cache before being passed to the application for transmission, in which case the copy step is obviously inefficient. There are two reasons for the copy step:

1. To ensure the data is in pinned down kernel memory during the time that the NIC is copying the data in or out. This applies when data is being transmitted or received and also has the benefit that the application is not able to tamper or damage the buffer following kernel protocol processing.

2. In the case of transmission it is often the case that the send() system call returns successfully before the data has been actually transmitted. The copy step enables the OS to retain the data in case retransmission is required by the protocol layer.

[0021] Even if the copy step were omitted, on data reception, a cache load would take place when the application accessed the kernel buffer. Many people have recognized (see, e.g. U.S. Pat. No. 6,246,683) that these additional copies have been the cause of performance degradation. However, the solutions presented so far have all involved some excess data movement. It would be desirable to reduce this overhead. The inventors of the present invention have recognised that an overlooked problem is not the copying, but the user to kernel context switching and interrupt handling overheads. U.S. Pat. No. 6,246,683, for instance, does nothing to avoid these overheads.

[0022] During a context switch on a general purpose operating system many registers have to be saved and restored, and TLB entries and caches may be flushed. Modern processors are heavily optimized for sustained operation from caches and architectural constraints (such as the memory gap) are such that performance in the face of large numbers of context switches is massively degraded. Further discussion of this is given in "Piglet: A Low-intrusion Vertical Operating System", S. J. Muir and J. M. Smith, Tech. rep. MS-CIS-00-04, Univ. of PA, Jan. 2000. Hence it would be desirable to reduce context switches during both data transfer and connection management.

[0023] In order to remove the cost of context switches from data transmission and reception, VIA (Virtual Interface Architecture) was developed as an open standard from academic work in U-NET. Further information is available in the Virtual Interface Architecture Specification available from www.vidf.org. Some commercial implementations were made and it has since evolved into the Infiniband standard. The basic principle of this system is to enhance the network interface hardware to provide each application (network endpoint) with its own pair of DMA queues. T_x and R_x). The architecture comprises a kernel agent, a hardware NIC, and a user/application level interface. Each application at the user level is given control of a VI (Virtual Interface). This comprises two queues, one for transmission, one for reception (and an optional CQ completion queue). To transmit some data on a VI, the application must:
1. Ensure data is in a buffer which has been pinned down. (This would require a system call to allocate)
2. Construct a descriptor which contains a buffer pointer and length and add a pointer to the descriptor onto the send queue. (N.B. the descriptor must also be in pinned down memory).
3. If necessary the application indicates that the work queue is active by writing to a hardware "doorbell" location on the NIC which is associated with the VI endpoint.
4. At some time later the NIC processes the send queue and DMAs the data from the buffer, forms a network packet and transmits to the receiver. The NIC will then

mark the descriptor associated with the buffer to indicate that the data has been sent.

[0024] It is possible to also associate the VI with a completion queue. If this has been done the NIC will post an event to the completion queue to indicate that the buffer has been sent. Note that this is to enable one application to manage a number of VI queues by looking at the events on only one completion queue.

[0025] Either the VI send queue or completion queue may be enabled for interrupts.

[0026] To receive a buffer the application must create a descriptor which points to a free buffer and place the descriptor on the receive queue. It may also write to the "doorbell" to indicate that the receive queue is active.

[0027] When the NIC has a packet which is addressed to a particular VI receive queue, it reads the descriptor from the queue and determines the Rx buffer location. The NIC then DMAs the data to the receive buffer and indicates reception by:

1. marking the descriptor
2. generating an event on the completion queue (if one has been associated with the VI)
3. generating an interrupt if one has been requested by the application or kernel (which marks either the R_x queue or completion queue).

[0028] There are problems with this queue based model:

1. Performance for small messages is poor because of descriptor overheads.
2. flow control must be done by the application to avoid receive buffer overrun.

[0029] Also, VIA does not avoid context switches for connection setup and has no error recovery capabilities. This is because it was intended to be used within a cluster where there are long-lived connections and few errors. If an error occurs, the VI connection is simply put into an error state (and usually has to be torn down and recreated).

9

[0030] VIA connection setup and tear down proceeds using the kernel agent in exactly the same manner as described for kernel stack processing. Hence operations such as Open, Connect, Accept etc all require context switches into the kernel. Thus in an environment where connections are short lived (e.g. WWW) or errors are frequent (e.g. Ethernet) the VIA interface performs badly.

[0031] VIA represents an evolution of the message passing interface, allowing user level access to hardware. There has also been another track of developments supporting a shared memory interface. Much of this research was targeted at building large single operating system NUMA (non-uniform memory architecture) machines (e.g. Stanford DASH) where a large supercomputer is built from a number of processors, each with local memory, and a high-speed interconnect. For such machines, coherency between the memory of each node was maintained by the hardware (interconnect). Coherency must generally ensure that a store/load operation on CPU_1 will return the correct value even where there is an intervening store operation on CPU_2. This is difficult to achieve when CPU_1 is allowed to cache the contents of the initial store and would be expected to return the cached copy if an intervening write had not occurred. A large part of the IEEE standard for SCI (Scalable Coherent Interconnect) is taken up with ensuring coherency. The standard is available from www.vesa.org.

[0032] Because of the NUMA and coherency heritage of shared memory interconnects, the management and failure modes of the cluster were that of a single machine. For example implementations often assumed.

1. Single network address space which was managed by a trusted cluster-wide service (e.g. DEC's Memory Channel: see R Gillett Memory Channel Network for PCI, IEEE Micro 16(2), 12-18 Feb. 96)

2. No protection (or no page level protection) measures to protect local host memory from incoming network access (e.g. SCI)

3. Failure of a single processor node causing failure of whole "machine" (e.g. SCI)

[0033] In a Memory Channel implementation of a cluster wide connection service, physically, all network writes are passed to all nodes in the cluster at the same time, so a receiving node just matches writes against its incoming window. This method does provide incoming protection (as we do), but address space management requires communication with the management node and is inefficient.

[0034] SCI is similar except that 12 bits of the address space is dedicated to a node identifier so that writes can be directed to a particular host. The remaining 48 bits is host implementation dependent. Most implementations simply allocate a single large segment of local memory and use the 48 bits as an offset.

[0035] No implementation has addressed ports or distributed connection management as part of its architecture.

[0036] Some implementations provide an event mechanism, where an event message can be sent from one host to another (or from network to host). When these are software programmable, distributed connection set up using ports is possible. However, since these mechanisms are designed for (rare) error handling (e.g. where a cable is unplugged), the event queue is designed to be a kernel only object--hence context switches are still required for connection management in the same manner as the VIA or kernel stack models.

BRIEF SUMMARY OF THE INVENTION

[0037] According to one aspect of the present invention there is provided a communication interface for providing an interface between a data link and a data processor, the data processor being capable of supporting an operating system and a user application, the communication interface being arranged to: support a first queue of data received over the link and addressed to a logical data port associated with a user application; support a second queue of data received over the link and identified as being directed to the operating system; and analyse data received over the link and identified as being directed to the operating system or the data port to determine whether that data

meets one or more predefined criteria, and if it does meet the criteria transmit an interrupt to the operating system.

[0038] Conveniently the user application has an address space and the first queue is located in that address space. Conveniently the operating system has an address space and the second queue is located in that address space. Most conveniently at least part of the address space of the user application is the same as at least part of the address space of the operating system. Preferably the all address space of the user application lies within the address space of the operating system.

[0039] The communication interface is preferably arranged apply to the first queue data received over the link and identified as being directed to the data port. The communication interface is preferably arranged apply to the second queue data received over the link and identified as being directed to the operating system.

[0040] Preferably one of the predefined criteria is such that if the data received over the link matches one or more predetermined message forms then the communication interface will transmit an interrupt to the operating system.

[0041] Preferably the communication interface is arranged to, if the data meets one or more of the predefined criteria and one or more additional criteria transmit an interrupt to the operating system and transmit a message to the operating system indicating a port to which the data was addressed. Preferably the additional criteria are indicative of an error condition.

[0042] Preferably the communication interface is arranged to support a third queue of data received over the link and addressed to a logical data port associated with a user application, and is arranged to apply to the first queue data units received over the link and of a form having a fixed length and to apply to the third queue data units received over the link and of a form having a variable length. Preferably the data units of a fixed size include messages received over the link and interpreted by the communication

12

interface as indicating an error status. Preferably the data units of a fixed size include or may include messages received over the link and interpreted by the communication interface as indicating a request for or acknowledgement of set-up of a connection. Preferably the data units of a fixed size include messages received over the link and interpreted by the communication interface as indicating a data delivery event.

[0043] Preferably the communication interface is arranged to analyse the content of each data unit received over the link and to determine in dependence on the content of that data unit which of the said queues to apply the data unit to.

[0044] Preferably the communication interface is configurable by the operating system to set the said criteria.

[0045] Preferably one or both of the communication interface and the operating system is responsive to a message of a predetermined type to return a message including information indicative of the status of the port.

[0046] According to the present invention there is also provided a communication system including a communication interface as set out above and the said data processor.

[0047]   The data processor is preferably arranged to, when the processing of an application with which a data port is associated is suspended, set the criteria such that the communication interface will transmit an interrupt to the operating system on receiving data identified as being directed to that data port.

[0048]   According to a second aspect of the present invention there is provided a communication interface for providing an interface between a data link and first data processing apparatus including a memory, the data interface being such that a region of the memory of the first data processing apparatus can be mapped on to memory of a second data processing apparatus connected to the communication interface by the link, the communication interface being arranged to, on establishing a mapping of a first range

13

of one or more memory locations in the second data processing apparatus on to a second range of one or more memory locations in the first data processing apparatus, transmit to the second data processing apparatus data identifying the first range of memory locations.

[0049]   Preferably the memory of the second data processing apparatus is virtual memory. Preferably the memory locations in the memory of the second data processing apparatus are virtual memory locations. Most preferably said one or more memory locations in the memory of the first data processing apparatus are one or more virtual memory locations and the communication interface is arranged to, on establishing the said mapping, establish a further mapping of the one or more virtual memory locations on to one or more physical memory locations in the memory of the first data processing apparatus

[0050]   Preferably the communication interface is arranged to, on establishing a mapping of a first range of one or more memory locations in the memory of the second data processing apparatus on to a second range of one or more memory locations in the memory of the first data processing apparatus, allocate an identity to that mapping and transmit that identity to the second data processing apparatus

[0051]   Preferably the communication interface is capable of communicating by means of data messages which specify a destination port to which data they contain is to be applied.

[0052]   Preferably the communication interface is arranged to, on establishing a mapping of a first range of one or more memory locations in the memory of the second data processing apparatus on to a second range of one or more memory locations in the memory of the first data processing apparatus, determine check data and transmit the check data to the second data processing apparatus, and wherein the communication interface is arranged to reject subsequent communications over the mapping which do not indicate the check data. Preferably the check data is randomly generated by the

14

communication interface. Conveniently, to indicate the check data a communication includes the check data.

[0053]   Preferably the communication interface is arranged to modify the check data, according to a predefined scheme, during the operation of the mapping. Then it is subsequent communications over the mapping that do not indicate that modified data that the communication interface is preferably arranged to reject. Preferably the check data represents a number and the predefined scheme is to increment the number represented by the check data by a predefined amount each time a predefined number of communications over the mapping is accepted. The predefined amount is preferably but not necessarily one. The predefined number is preferably but not necessarily one.

[0054]   Preferably the communication interface is arranged to reject subsequent communications over the mapping which indicate a request for accessing data outside the first range.

[0055]   According to the present invention there is also provided a communication system including a communication interface as set out above and the said data processor.

[0056]   The data processor is preferably capable of supporting an operating system and a user application. The system preferably comprises a data store which stores items of data defining operation parameters for communications over the data link to transmit data stored in the first range or receive data for storage in the first range.

[0057]   The operating system may be arranged to permit a user application to access one or more items of data in the data store dependent on a level of trust granted to the application.

[0058]   The check data is preferably stored as one of the items of data in the data store, the operating system is arranged to permit at least some user applications to have write access to that item of data, and the communication interface is arranged to, in order

15

to determine the check data, read the content of that item of data and treat it as the check data.

[0059]   Preferably items of data in the data store define the start and end points of the first range of memory locations in the memory of the first data processing apparatus and store the start and end points of the second range of memory locations in the memory of the second data processing apparatus. Preferably the operating system is arranged to permit applications having one or more levels of trust to have write access to the items of data in the data store that second, and store the start and end points of the second range of memory locations in the memory of the second data processing apparatus and to permit no applications to have write access to the items of data in the data store that define the start and end points of the first range of memory locations in the memory of the first data processing apparatus.

[0060]   The present invention will now be described by way of example with reference to the accompanying drawings.

[0061]   Preferably the communication interface is capable of supporting a plurality of mappings each of a respective first range of one or more virtual memory locations in the second data processing apparatus on to a respective second range of one or more memory locations in the first data processing apparatus, and for each such mapping a respective further mapping of the respective one or more virtual memory locations on to one or more physical memory locations in the memory of the first data processing apparatus.

[0062]   Preferably the communication interface includes a translation interface for translating accesses to or from each of the said ranges of one or more virtual memory locations into accesses to or from the respective one or more physical memory locations in the memory of the first data processing apparatus and for translating accesses to or from each of the one or more physical memory locations in the memory of the first data processing apparatus into accesses to or from the respective ranges of one or more virtual memory locations. Preferably the virtual memory locations are local bus addresses, for

example PCI bus addresses. An access to a location is suitably a write access to that location. An access from a location is suitably a read access from that location.

[0063]   Preferably the communication interface comprises a mapping memory arranged to store specifications of the said further mappings. The mapping memory preferably comprises a first mapping memory local to the translation interface, and a second mapping memory less local to the translation interface than the first mapping memory, and wherein the communication interface is arranged to store specifications of all of the further mappings in the second mapping memory, and to store specifications of only some of the further mappings in the first mapping memory. Preferably the first mapping memory is an associative memory.

[0064]   Preferably the translation interface is arranged to, in order to translate between an access to or from one of the said ranges of one or more virtual memory locations and an access to or from the respective one or more physical memory locations in the memory of the first data processing apparatus, preferentially access the first mapping memory to implement the translation, and if the specification of the mapping of the range of virtual memory locations the subject of the access is not stored in the first mapping memory to access the second mapping memory to implement the translation.

[0065]   Preferably the communication interface is arranged to store specifications of the most recently used further mappings in the first mapping memory. Preferably it is arranged to, if an attempt to access a specification from the first mapping memory is unsuccessful, replace a specification in the first mapping memory with the specification the attempt to access which was unsuccessful.

BRIEF DESCRIPTION OF THE SEVERAL VIEWS OF THE DRAWINGS

[0066]   In the drawings:

[0067]   FIG. 1 illustrates mapping of one address space on to another over a network;

17

[0068]   FIG. 2 illustrates the architecture of a prior art memory mapped architecture;

[0069]   FIG. 3 is a schematic diagram of a data transmission system;

[0070]   FIGS. 4 and 5 illustrate mapping of bits of an address;

[0071]   FIG. 6 illustrates memory space apertures and their management domains;

[0072]   FIG. 7 illustrates features of a port;

[0073]   FIG. 8 illustrates a queue with control blocks;

[0074]   FIG. 9 illustrates a dual queue mechanism;

[0075]   FIG. 10 shows an example of an outgoing aperture table;

[0076]   FIG. 11 shows an example of an incoming aperture table;

[0077]   FIG. 12 shows the steps in a PCI write for an outgoing aperture; and

[0078]   FIG. 13 illustrates the operation of pointers in fixed and variable length queues.

DETAILED DESCRIPTION OF THE INVENTION

[0079]   FIG. 3 is a schematic diagram of a data transmission system whereby a first data processing unit (DPU) 20 can communicate with a second data processing unit 21 over a network link 22. Each data processing unit comprises a CPU 23, 24 which is connected via a memory bus 25, 26 to a PCI controller 27, 28. The PCI controllers control communications over respective PCI buses 29,30, to which are connected NICs

18

31, 32. The NICs are connected to each other over the network. Other similar data processing units can be connected to the network to allow them to communicate with each other and with the DPUs 20, 21. Local random access memory (RAM) 33, 34 is connected to each memory bus 25, 26.

[0080]   The data transmission system described herein implements several significant features: (1) dynamic caching of aperture mappings between the NICs 31, 32; (2) a packet oriented setup and teardown arrangement for communication between the NICs; and (3) the use of certain bits that are herein termed "nonce bits" in the address space of one or both NICs.

[0081]   Dynamic Caching of Aperture Entries

[0082]   A small number of aperture mappings can be stored efficiently using a static table. To implement this, a number of bits (the map bits) of an address are caught by the address decode logic of an NIC and are used as an index into an array of memory which contains the bits that are used for reversing the mapping (the remap bits). For example, in a system of the type illustrated in FIG. 3 an NIC might receive over the PCI bus 29 a request for reading or writing data at a specified local address. The NIC stores a mapping that indicates the remote address that corresponds to that local address, the transformation being performed by substituting one or more of the bits of the local address. For example, the second and third nibbles of the address could be substituted. In that case to access the remote address that corresponds to a local address of 0x8210BEEC the NIC would access the mapping table, determine the mapping for bits "21" (suppose that is bits "32") and then address the corresponding remote address (in this example 0x8320BEEC). (See FIG. 4)

[0083]   This method is scalable up to a few hundred or thousand entries depending on the implementation technology used (typically FPGA or ASIC) but is limited by the space available within the device that is used to hold the mapping table. A superior method of implementation is to store the mappings in a larger store (to which access is

19

consequently slower) and to cache the most recently used mappings in an associative memory that can be accessed quickly. If a match for the bits that are to be substituted is found in the associative memory (by a hardware search operation) then the remap is made very quickly. If no match is found the hardware must perform a secondary lookup in the larger memory (in either a table or tree structure). Typically the associative memory will be implemented on the processing chip of the NIC, and the larger memory will be implemented off-chip, for example in DRAM. This is illustrated in FIG. 5. This method is somewhat similar to the operation of a TLB on a CPU; however here it is used for an entirely different function: i.e. for the purpose of aperture mapping on a memory mapped network card.

[0084]   In practice, the mapping information must contain all the address information required to transmit a packet over a network. This is discussed in more detail below.

[0085]   Packet Oriented Connection Setup and Tear Down Protocol

[0086]   A protocol will now be described for establishing a connection between two applications' address spaces using apertures, where there are two administration domains (one belonging to each of the communicating hosts). The general arrangement is illustrated in FIG. 6. In domain A there is a host A having a virtual address space A and an NIC A that can access the virtual address space. In domain B there is a host B having a virtual address space B and an NIC B that can access the virtual address space. The NICs are connected together over a network.

[0087]   In this example mapping entries for devices in domain A can only be set by the operating system on host A. A further implementation in which an application A running on host A is allowed to set some (but not all) bits on an aperture mapping within domain A is described below.

[0088]   The connection protocol to be described uses IP (Internet Protocol) datagrams to transfer packets from one host to another (just as for standard Ethernet networks). The

datagrams are addressed as <host:port> where <host> is the network identifier of the destination host and <port> is an identifier for the application (NB each application may have a number of allocated parts corresponding to different network connections) within the host. It will be appreciated that the present protocol could be used over other transport protocols than IP.

[0089]   In the present protocol the connection setup proceeds as follows, assuming host A wishes to make an active connection to a passive (accepting) host B on which an application B is running.

1. Application B publishes its accepting internet address $<host_B:port_B>$ this can be accessed over the network in the normal way.

2. Application A (which for convenience will be referred to as host A) presents a request to Operating System A for the creation of an incoming aperture onto memory within host A to be used for communication. Once this aperture has been defined its details are programmed on NIC A so that incoming network writes that are directed to addresses in that virtual space will be directed onto the corresponding real addresses in memory A. The aperture will be given a reference address: in-index A.

3. The host A sends an IP datagram to $<host_B:port_B>$ which contains: the connect message:

    $[CONNECT/in\text{-}index_A]$

    Note that the full IP datagram will also contain source and destination IP addresses (and ports), as normal.

4. The connect message is received by application B. The message may be received either directly to user level or to the operating system (according to the status of the dual event queue) as described later.

5. Host B recognises the message as being a request to connect to B, offering the aperture in-index A. Using rules pre-programmed at B (typically for security reasons) host B will decide whether to reject or accept the connection. If B decides to accept the connection, it creates an (or uses a pre-created) incoming aperture which is mapped onto memory B and is given reference address: in-

index$_B$. Host B may choose to create a new port for the connection: port'$_B$. Host B sends back to host A an accept message as an IP datagram:

[ACCEPT/:port'$_B$/in-index$_B$]

to host A. Note that the full IP datagram will also contain source and destination IP addresses (and ports), as normal.

Once this has been received, each host has created an aperture, each NIC is set up to perform the mapping for requests to read or write in that aperture, and each host knows the reference address of the other host's aperture.

6. Following the messaging discussed so far, both hosts create outgoing apertures. A creates an aperture which maps application A's virtual address space onto NIC$_A$ outgoing aperture OUT_index$_A$. This outgoing aperture maps onto [host$_B$:in-index$_B$] which maps onto memory B. Host B creates a similar outgoing aperture out-index$_B$ which maps onto memory A. By this means, bi-directional communication is possible through the memory mapped regions. At any time the applications may send a message to the port, which is associated with the memory mapping. These may be used to guarantee out of band data for example:

    (a)   A CLOSE message to indicate that the connection and hence memory mappings should be closed down

    (b)   An ALIVE message to request a response from an non-responding application [ALIVEACK would be the response]

    (c)   An ERROR message which us generated by any hardware element on the data path which has detected a data transfer error. This message is important because it allows feedback to be provided from the memory mapped interface.

Note that where an application already has a virtual address mapping onto an outgoing aperture, step 6 reduces to a request for the NIC to map the outgoing aperture onto a particular host's incoming aperture. This is described further in terms of user level connection management below.

[0090]   Dual Event Queues

[0091]   In the present context a port will be considered to be an operating system specific entity which is bound to an application, has an address code, and can receive messages. This concept is illustrated in FIG. 7. One or more incoming messages that are addressed to a port form a message queue, which is handled by the operating system. The operating system has previously stored a binding between that port and an application running on the operating system. Messages in the message queue for a port are processed by the operating system and provided by the operating system to the application to which that port is bound. The operating system can store multiple bindings of ports to applications so that incoming messages, by specifying the appropriate port, can be applied to the appropriate application.

[0092]   The port exists within the operating system so that messages can be received and securely handled no matter what the state of the corresponding application. It is bound (tethered) to a particular application and has a message queue attached. In traditional protocol stacks, e.g. in-kernel TCP/IP all data is normally enqueued on the port message queue before it is read by the application. (This overhead can be avoided by the memory mapped data transfer mechanism described herein).

[0093]   In the scheme to be described herein, only out of band data is enqueued on the port message queue. FIG. 7 illustrates this for a CONNECT message. In FIG. 7, an incoming packet E, containing a specification of a destination host and port (field 50), a message type (field 51) and an index (field 52), is received by NIC 53. Since this data is a CONNECT message it falls into the class of out of band data. However, it is still applied to the message queue 54 of the appropriate port 55, from where it can be read by the application that has been assigned by the operating system to that port.

[0094]   A further enhancement is to use a dual queue, associated with a port. This can help to minimise the requirements to make system calls when reading out of band messages. This is particularly useful where there are many messages e.g. high connection rate as for a web server, or a high error rate which may be expected for Ethernet.

[0095]   At the beginning of its operations, the operating system creates a queue to handle out of band messages. This queue may be written to by the NIC and may have an interrupt associated with it. When an application binds to a port, the operating system creates the port and associates it with the application. It also creates a queue to handle out of band messages for that port only. That out of band message queue for the port is then memory mapped into the application's virtual address space such that it may de-queue events without requiring a kernel context switch.

[0096]   The event queues are registered with the NIC, and there is a control block on the NIC associated with each queue (and mapped into either or both the OS or application's address space(s)).

[0097]   A queue with control blocks is illustrated in FIG. 8. The queue 59 is stored in memory 60, to which the NIC 61 has access. Associated with the queue are a read pointer (RDPTR) 62a and a write pointer (WRPTR) 63a, which indicate the points in the queue at which data is to be read and written next. Pointer 62a is stored in memory 60. Pointer 63a is stored in NIC 61. Mapped copies of the pointers: RDPTR' 62b and WPTR' 63b are stored in the other of the NIC and the memory than the original pointers. In the operation of the system:

1. The NIC can determine the space available for writing by comparing RDPTR' and WRPTR, which it stores locally.
2. NIC generates out of band data when it is received in a datagram and writes it to the queue 59.
3. The NIC updates WRPTR and WRPTR' when the data has been written, so that the next data will be written after the last data.
4. The application determines the space available for reading by comparing RDPTR and WRPTR' as access from memory 60.
5. The application reads the out of band data from queue 59 and processes the messages.
6. The application updates RDPTR and RDPTR'.

7. If the application requires an interrupt, then it (or the operating system on its behalf) sets the IRQ 65a and IRQ' 65b bits of the control block 64. The control block is stored in memory 60 and is mapped onto corresponding storage in the NIC. If set, then the NIC would also generate an interrupt on step 3.

[0098] If an interrupt is generated, then firstly the PCI interrupt line is asserted to ensure the computer's interrupt handler is executed, but also a second message is delivered into the operating system's queue. In general, this queue can handle many interrupt types, such as hardware failure, but in this case, the OS queue contains the following message [ODBDATA:PORT] indicating that out of band data has been delivered to the application queue belonging to [PORT]. The OS can examine the data in queue 59 and take appropriate action. The usual situation will be that the application is blocked or descheduled and the OS must wake it (mark as runnable to the scheduler).

[0099] This dual queue mechanism enables out of band data to be handled by the application without involving the OS--while the application is running. Where the application(s) is blocked, the second queue and interrupt enable the OS to determine which of potentially many application queues have had data delivered. The overall arrangement is illustrated in FIG. 9.

[0100] The out of band (OOB) queue holds out of band data, which are:
1. Error events associated with the port
2. Connection setup messages and other signalling messages from the network and other applications
3. Data delivery events, which may be generated either by the sending application the NIC or the receiving OS.

[0101] If the queue is to contain variable sized data then the size of the data part of each message must be included at the start of the message.

[0102]   When applications are to communicate in the present system over shared memory, a single work queue can be shared between two communicating endpoints using non-coherent shared memory. As data is written into the queue, write pointer (WRPTR) updates are also written by the transmitting application into the remote network-mapped memory to indicate the data valid for reading. As data is removed from the queue, read pointer (RDPR) updates are written by the receiving application back over the network to indicate free space in the queue.

[0103]   These pointer updates are conservative and may lag the reading or writing of data by a short time, but means that a transmitter will not initiate a network transfer of data until buffer is available at the receiver, and the low latency of the pointer updates means that the amount of queue buffer space required to support a pair of communicating endpoints is small. The event mechanism described above can be used to allow applications to block on full/empty queues and to manage large numbers of queues via a multiplexed event stream, which is scalable in terms of CPU usage and response time.

[0104]   Variable length data destined for an event queue would be delivered to a second queue. This has the advantage of simplifying the event generation mechanism in hardware. Thus the fixed size queue contains simple events and pointers (size) into the variable length queue

1. As shown in FIG. 13, the difference between RDPTR, and WRPTR, indicates the valid events in the queue, and also the number of events because they are of fixed size.
2. The event Var 10 (for illustration) indicates that a variable sized event of size 10 words has been placed on the variable sized queue.
3. The difference between $WRPTR_2$ and $RDPTR_2$ indicates only the number of words which are in the variable sized queue, but the application is able to dequeue the first event in its entirety by removing 10 words.
4. The application indicates processing of an event to the NIC by updating the RDPTR on the NIC's memory

(a) for the static queue by the number of events processed multiplied by the size of each event

(b) for the variable sized queue by the number of words consumed (i.e. the same for both cases)

5. The data on the variable length queue may also contain the size (e.g. if it is a UDP/IP packet)

[0105]   Enhanced Aperture Mappings and "Nonce Bits"

[0106] In this implementation, additional bits, termed "nonce bits" are provided in order to protect against malfunctioning or malicious hardware or software writing inadvertently to apertures. To illustrate this, the following network mapping will be discussed:

<virtual memory address> → <PCI address> → <host:in-index> → …

…<network packet> → <PCI address> → <physical memory address> → …

…<virtual memory address>

[0107] When performing the mapping to <host in-index> the NIC is able to create an outgoing packet which is addressed by <host: in-index>. This will be recognized by the NIC that receives the packet as being a packet intended for processing as an aperture packet, rather than as a packet intended to pass via a port to a corresponding application. Thus the packet is to be presented to the incoming aperture lookup hardware.

[0108] It should first be noted that under the scheme described above, the PCI address to which the data is sent encodes both the aperture mapping and an offset within the aperture. This is because the NIC can form the destination address as a function of the address to which the message on the PCI bus was formed. The address received by the NIC over the PCI bus can be considered to be formed of (say) 32 bits which include an aperture definition and a definition of an offset in that aperture. The offset bits are also encoded in the outgoing packet to enable the receiving NIC to write the data relative to the incoming aperture base. In the case of a data write the resulting network packet can be considered to comprise data together with a location definition comprising an offset,

an in-index and an indication of the host to which it is addressed. At the receiving NIC at the host this will be considered as instructing writing of the data to the PCI address that corresponds to that aperture, offset by the received offset. In the case of a read request the analogous operation occurs. This feature enables an aperture to be utilized as a circular queue (as described previously) between the applications and avoids the requirement to create a new aperture for each new receive data buffer.

[0109] In this implementation the network packet also contains the nonce bits. These are programmed into the aperture mapping during connection setup and are intended to provide additional security, enabling apertures to be reused safely for many connections to different hosts.

[0110] The processing of the nonce bits for communications between hosts A and B is as follows:

1. At host A a random number is selected as nonce A.
2. Nonce A is stored in conjunction with an aperture in-index A
3. A connect message is sent to host B to set up communications in the way generally as described above. In this example the message also includes nonce A. Thus the connect message includes port B, in-index A, nonce A.
4. On receiving the connect message host B stores in-index A and nonce A in conjunction with outgoing aperture B.
5. Host B selects a random number as nonce B
6. Nonce B is stored in conjunction with an aperture in-index B
7. An accept message is sent to host B to accept the set up of communications in the way generally as described above. In this example the message also includes nonce B. Thus the accept message includes port B', in-index B, nonce B.
8. Host A stores in-index B and nonce B in conjunction with outgoing aperture A.

[0111] Once the connection is set up to include the nonce bits all packets sent from A to B via outgoing aperture$_A$ will contain nonce B. When received the NIC$_B$ will look up in-index B and compare the received nonce value with that programmed at B. If they differ, the packet is rejected. This is very useful if a malfunctioning application holds onto a stale connection: it may transmit a packet which has a valid [host:in-index] address, but would have old nonce bits, and so would be rejected.

[0112] Remembering that the user level application has a control block for the out of band queue, this control block can also be used to allow control of the apertures associated with the application, in such a way that connection setup and tear down may be performed entirely at user level.

[0113] Note that some parts of the aperture control block only are user programmable, others must only be programmed by the operating system.

- User Programmable bits include: nonce bits, destination host bits
- O/System Programmable bits include:
  a) base address of incoming aperture (this prevents an application from corrupting memory buffers by mistake or malintent)
  b) source host bits (this prevents an application from masquerading as originating from another host).

[0114] For an untrusted application, kernel connection management would be performed. This means that out of band data would be processed only in the kernel, and no programmable bits would be made available to the application.

[0115] An example of an outgoing aperture table is shown in FIG. 10. Each row of the table represents an aperture and indicates the attributes of that aperture. It should be noted that:

1. A number of aperture sizes may be supported. These will be grouped such that the base address also gives the size of the aperture. Alternatively, a size field can be included in the aperture table.

2. The type field indicates the Ethernet type to use for the outgoing packet. It also indicates whether the destination address is a 4 byte IPv4 address or a 16 bit cluster address. (IPv6 addresses or other protocol addresses could equally be accommodated) The type field also distinguishes between event and data packets within the cluster. (An event packet will result in a fixed size event message appearing on the destinations event queue).

3. The PCI base address is OS programmable only, other fields may be programmed by the application at user level depending on the system's security policy.

4. Source Ethernet address, Source IP and Cluster address and possibly other information is common to all entries and stored in per NIC memory.

5. In all cases addressing of the outgoing Ethernet packet is either

                           \<Ethernet MAC\>\<IP host: IP port\> (in the case of a TCP/IP packet)

or

                           \<Ethernet MAC\>\<CI host: CI in-index: CI nonce: CI aperture offset\> (in the case of a CI (computer interface) packet)

(n.b. the offset is derived from the PCI address issued).

6. Each aperture is allocated an initial sequence number. This is incremented by the hardware as packets are processed and are optionally included in cluster address formats

[0116] An example of an incoming aperture table is shown in FIG. 11. Each row of the table represents an aperture and indicates the attributes of that aperture. The incoming aperture is essentially the reverse of the outgoing aperture. It should be noted that:

1. As well as the size being optionally encoded by having fixed size tables, the EthType can be optionally encoded by grouping separate aperture tables

2. The sequence number fields are optional and the receiver can set

    (a) whether sequence checking should be done

    (b) the value of the initial sequence number

If done this must also be communicated as part of the connection protocol, which could conveniently be performed in a similar way to the communication of nonce values from one host to another.

3. Similarly to outgoing apertures, some information is Per-NIC e.g. IP address, Ethernet address.

4. For application level robustness it is possible to "narrow" down an aperture by specifying an address and size which specifies a range which lies within the default range. This might be done when the application level data structure is of a size smaller, or different alignment, than the default aperture size and fine grained memory protection is required.

5. The map address is either the PCI address which the NIC should emit in order to write to memory for the aperture, or else a local (to the NIC's SRAM) pointer to the descriptor for the event queue.

[0117] A PCI write for an outgoing aperture is processed as shown in FIG. 12. The steps are as follows.

1. A PCI burst is emitted whose address falls within the range allocated to the NIC

2. The NIC's address decoder captures the burst and determines that the address is within the range of the apertures. (It could otherwise be a local control write).

3. Depending on the aperture size (which is coarsely determined from the address), the address is split into <base:offset>. E.g. for a 1 k aperture, the bottom 10 bits would be the offset. The base is fed into the aperture table cache to match the required packet header information.

4. Depending on the Ethernet packet type field either an IP/Ethernet or CI/Ethernet packet header is formed.

5. The CI packet would for instance, include the following fields:
   Data (containing the data payload of the PCI burst)
   Checksum (calculated by hardware over the contents of the header)
   Offset (by the address decoder)

Sequence number

Nonce

Aperture index

CI Host cluster address

6. If a number of PCI bursts arrive for a particular host, then they may be packed into a single Ethernet frame with compression techniques applied to remove redundant header information

7. In the present system a system-specific CRC or checksum is used to provide end-to-end protection and is appended to the data portion of the packet. Although the Ethernet packet also contains a CRC, it may be removed and recalculated on any hop (e.g. at a switch) and so does not provide protection against internal (e.g. switch-specific) corruptions.

8. If the sequence number is applied, then it is incremented and written back to the aperture table entry

[0118] For incoming packets, the reverse operation takes place. The incoming aperture is looked up and checked to be:

(a) valid;

(b) sequence number expected matches that of the packet;

(c) nonce matches (or port);

(d) expected Ethernet source address;

(e) expected IP or CI source addresses (which may be specified as a netmask to allow a range of source addresses to be matched);

[0119] Any one or more of these checks may be implemented or omitted, depending on the level of security required.

[0120] This lookup returns a field of: (base+extent) for the aperture. The offset is checked against the extent to ensure out of aperture access is not made and a PCI write is formed and emitted on the receiver's PCI bus with the format

... $DATA_2$ $DATA_1$	base + offset

[0121] If the PCI bus is stalled, (say on $DATA_N$) a new PCI transaction will be emitted

... $DATA_{N+1}$ $DATA_N$	base + offset + N

[0122] Similarly if consecutive CI data packets arrive they may be coalesced into larger PCI bursts simply by removing the redundant intermediate headers.

[0123] The applicant hereby discloses in isolation each individual feature described herein and any combination of two or more such features, to the extent that such features or combinations are capable of being carried out based on the present specification as a whole in the light of the common general knowledge of a person skilled in the art, irrespective of whether such features or combinations of features solve any problems disclosed herein, and without limitation to the scope of the claims. The applicant indicates that aspects of the present invention may consist of any such individual feature or combination of features. In view of the foregoing description it will be evident to a person skilled in the art that various modifications may be made within the scope of the invention.

1462790_1